Winning card games .... with 1000+ CPUs

Vincent Warmerdam & Bas Harenslak

Sushi Go!

SUSHI GO!
The Pick and Pass Card Game
2-5 Players
Ages 8+
GAMEWRIGHT
TEMPURA x2=5
WASABI NEXT NIGIRI x3
CHOPSTICKS SWAP FOR 2
SASHIMI x3=10
PUDDING END: MOST 6 LEAST -6

SUSHI GO PARTY!

SUSHI GO PARTY!

SUSHI GO PARTY!

SUSHI GO PARTY!

Naturally …

… this got me thinking.

I can use algorithms!

Naturally ...

This is where our story starts.

The goal:

1. Find a helpful algortihm
2. If need be, "borrow" my boss' creditcard for cloud resources.
3. Learn from it.
4. ?
5. Profit!

Towards a Computer Science Problem

This game can get quite deep, so keep it simple.

- I want to get better at the card game **myself** without the aid of a laptop. My girlfriend would justifiedly consider it cheating if I needed to consult the terminal at ever decision I need to make.
- So none of that Deep Reinforcement Voodoo.
- I know who I am, I'm bound to overengineer it and spend way too much time on it anyway.

Sushi Go!

SUSHI GO!
The Pick and Pass Card Game
2-5 Players
Ages 8+
GAMEWRIGHT
WASABI NEXT NIGIRI x3
TEMPURA x2=5
CHOPSTICKS SWAP FOR 2
SASHIMI x3=10
PUDDING END: MOST 6 LEAST -6

Sushi Go!

EGG NIGIRI 1
SALMON NIGIRI 2
SQUID NIGIRI 3
MAKI MOST:
TEMAKI MOST: 4 LEAST: -4
ONIGIRI UNIQUE: 1 4 9 16
SASHIMI x3=10
TEMPURA x2=5
TOFU x1=2 x2=6 x3+=0
CHOPSTICKS
URAMAKI
DUMPLING 1 3 6 10 15
EDAMAME 1 PER OPPONENTS WITH EDAMAME
EEL x1=-3 x2+=7
MISO SOUP 3 DISCARD IF ANY OTHERS PLAYED THIS TURN
MENU DRAW 4 CARDS, PLAY 1, RETURN THE REST 7
SOY SAUCE MOST COLORS 4
SPECIAL ORDER COPIES ANY CARD YOU HAVE PLAYED
SPOON ASK FOR ANY CARD. FIRST PLAYER TO LEFT WITH IT IN HAND SWAPS IT FOR THIS. 6
TAKE-OUT BOX FLIP ANY PLAYED CARDS. EACH ARE WORTH 2 10

What is the general order of importance for these cards?

```
cards = ["maki-1", "maki-2", "maki-3", "sashimi",
         "egg", "salmon", "squid", "wasabi", "pudding",
         "tempura", "dumpling", "tofu", "eel", "temaki"]
```

From Cards to Code

The code is pretty easy.

```python
def simulate(order, deck):
    random.shuffle(deck)
    hand_player, hand_opponent = give_hands(deck)
    random.shuffle(hand_opponent)
    table_player, table_opponent = [], []
    while len(hand_player) > 0:
        table_player.append(hand_opponent.pop())
        table_opponent.append(hand_player.pop())
        hand_player, hand_opponent = hand_opponent, hand_player
    return did_player_win(table_player, table_opponent)
```

This Problem, is a Problem.

The problem is big though:

$$14! = 87,178,291,200 \text{ combinations}$$

This Problem, is a Problem.

The problem is big though:

$$14! = 87,178,291,200 \text{ combinations}$$

It's is similar to the travelling salesman problem. The more cards I need to evaluate, the larger the search space.

I knew this problem was hard but technically, this implies that winning a cardgame from my girlfriend is **NP-HARD**.

This Problem, is a Problem.

The problem is big though:

$$14! = 87,178,291,200 \text{ combinations}$$

Before thinking attempting artificial intelligence via algorithms though, let's think consider domain knowledge and common sense.

This Problem, is a Problem.

```
cards = ["maki-1", "maki-2", "maki-3", "sashimi",
         "egg", "salmon", "squid", "wasabi", "pudding",
         "tempura", "dumpling", "tofu", "eel", "temaki"]
```

This Problem, is a Problem.

```
cards = ["maki-1", "maki-2", "maki-3", "sashimi",
         "egg", "salmon", "squid", "wasabi", "pudding",
         "tempura", "dumpling", "tofu", "eel", "temaki"]
```

From the game context, I know that

$$v(\text{maki-1}) < v(\text{maki-2}) < v(\text{maki-3})$$

$$v(\text{egg}) < v(\text{salmon}) < v(\text{squid})$$

This Problem, is a Problem.

This reduces the search space!

$$v(\text{maki-1}) < v(\text{maki-2}) < v(\text{maki-3})$$
$$v(\text{egg}) < v(\text{salmon}) < v(\text{squid})$$

Number of combinations now is $1/36$ of before:

$$\frac{14!}{3!3!} = 2,421,619,200$$

This Problem, is a Problem.

Number of combinations now is $1/36$ of before:

$$\frac{14!}{3!3!} = 2,421,619,200$$

From an algorithmic perspective, just thinking about the problem gave us a huge reduction in search space.

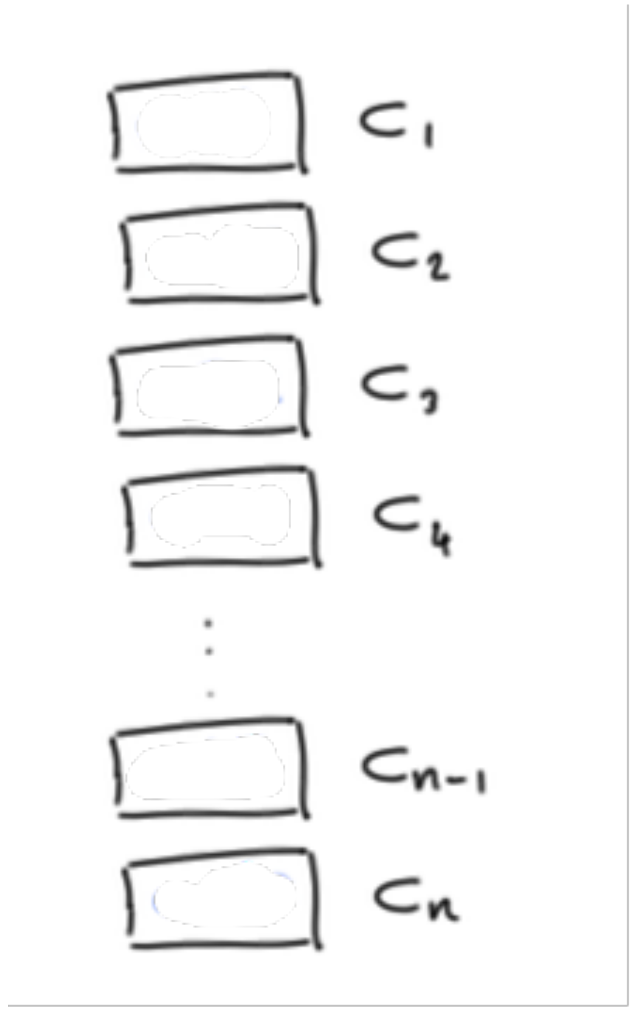
People should do this more often. Think before code!

This Problem, is a Problem.

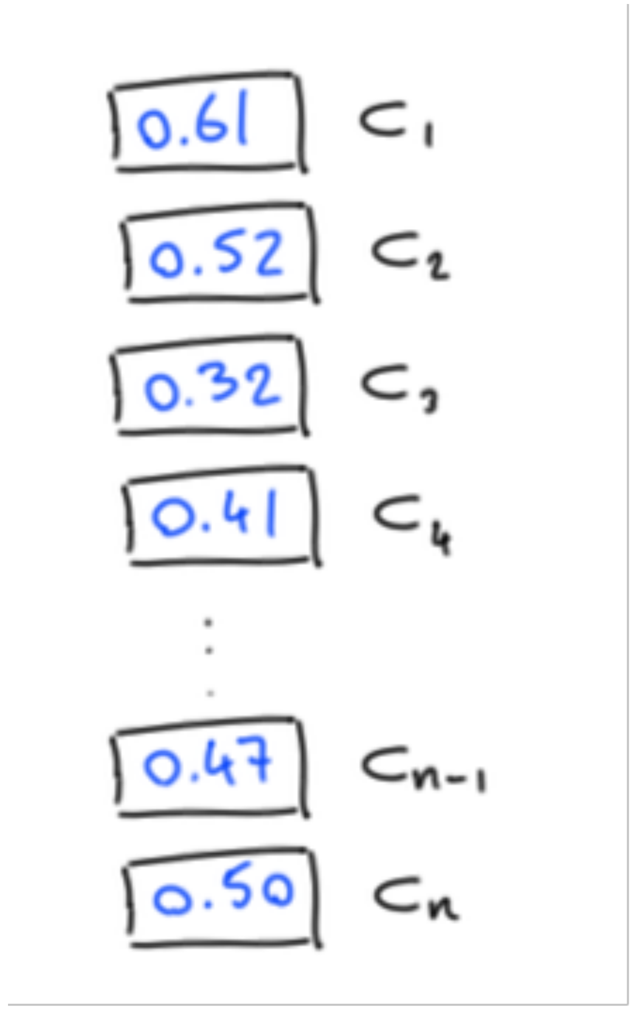
Even with the reduction, $2,421,619,200$ combinations is an not a small search space. What makes it worse: every combination needs plenty of simulations in order for it to be accurate.

So how would a algorithm work? The problem has many parallel parts …

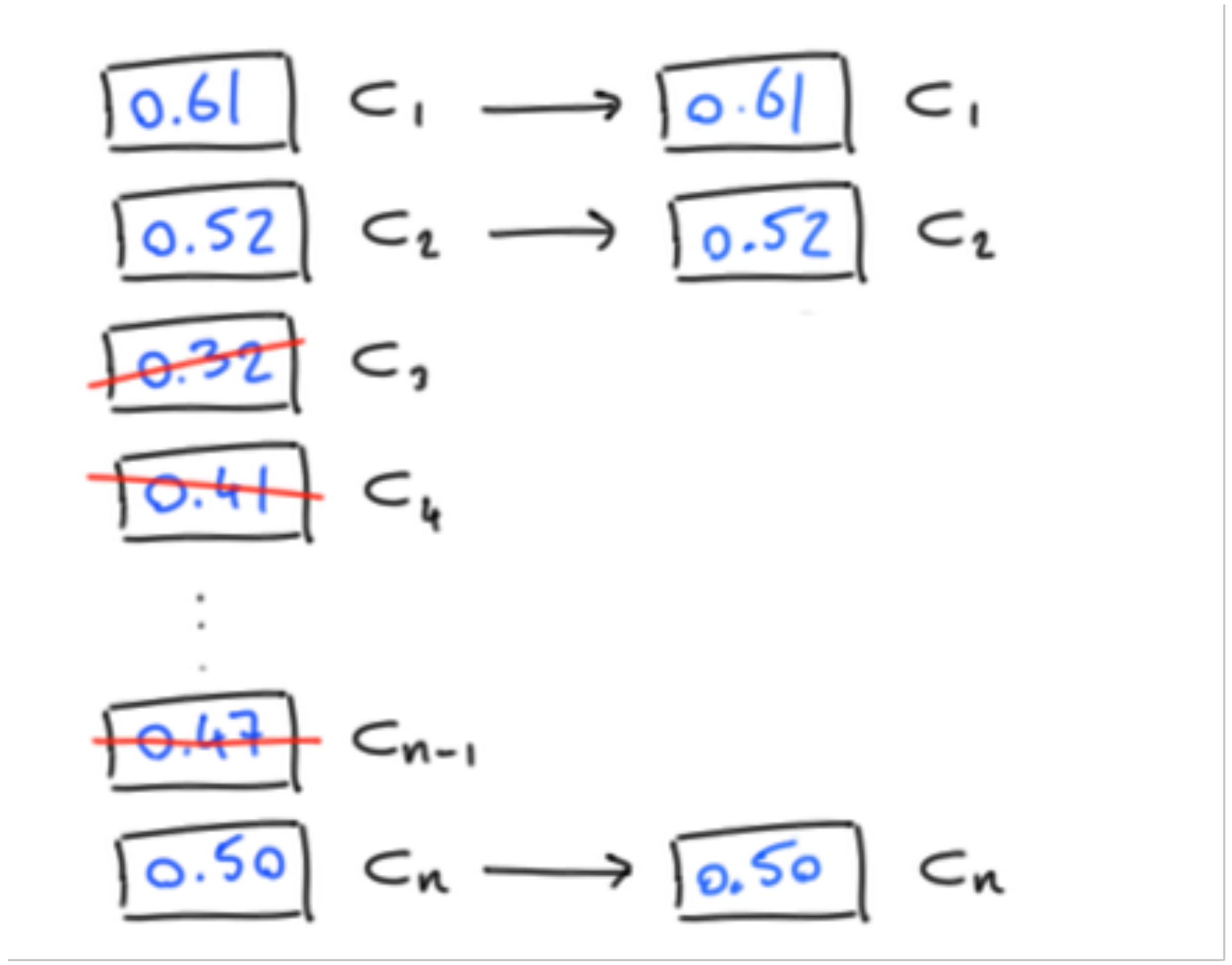
Idea of Evolutionary Heuristics.



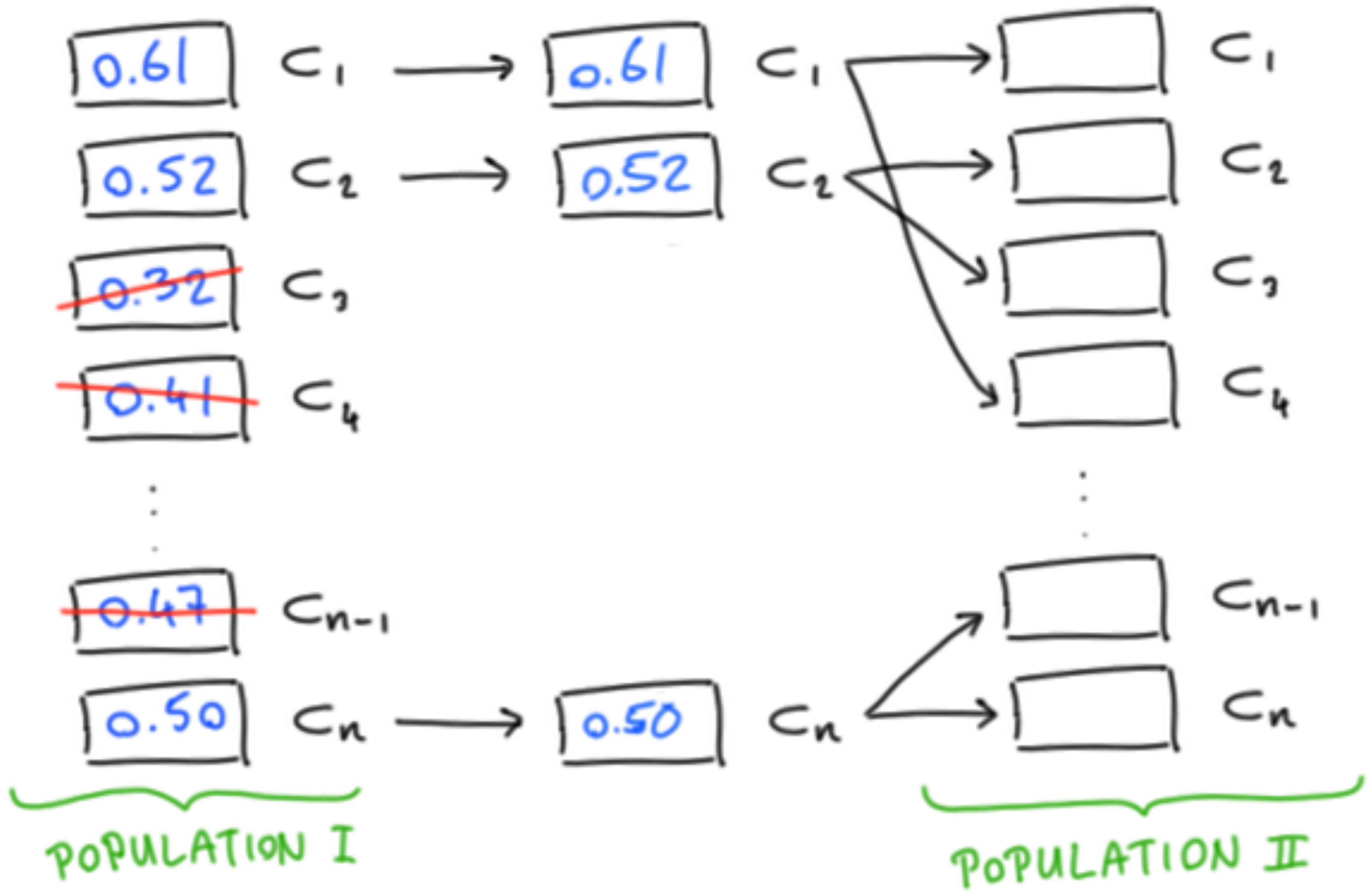
Idea of Evolutionary Heuristics.



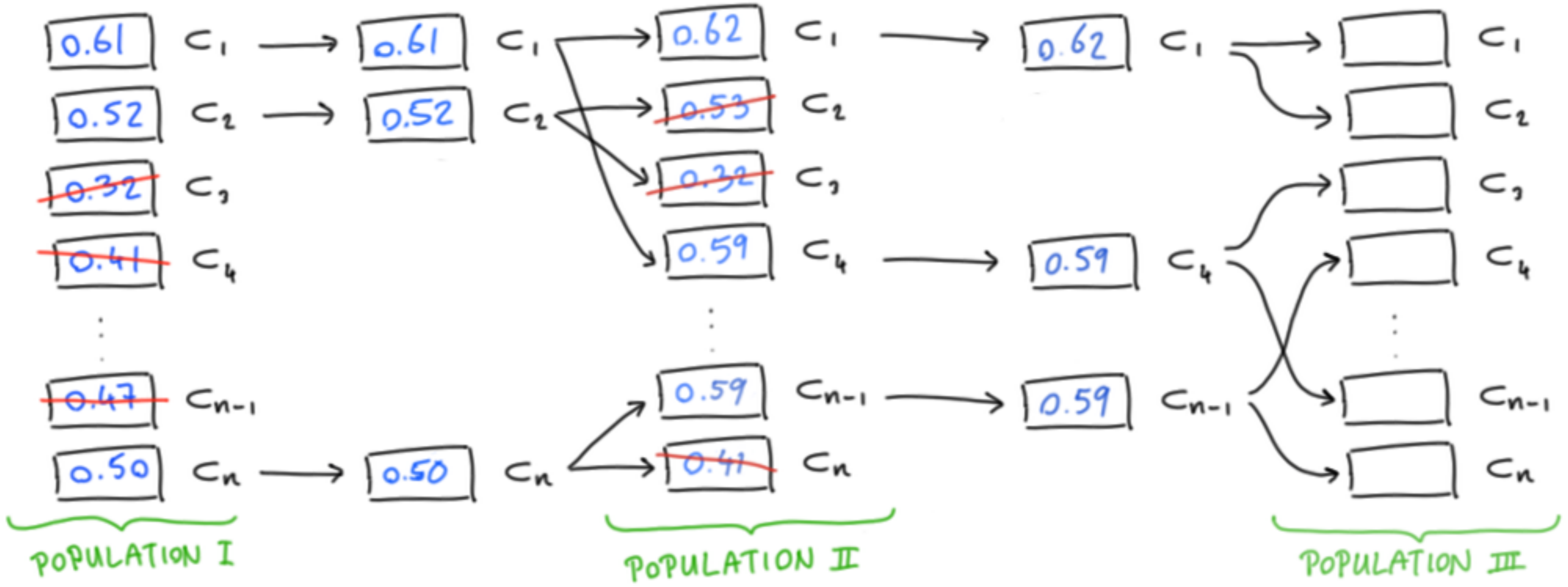
Idea of Evolutionary Heuristics.



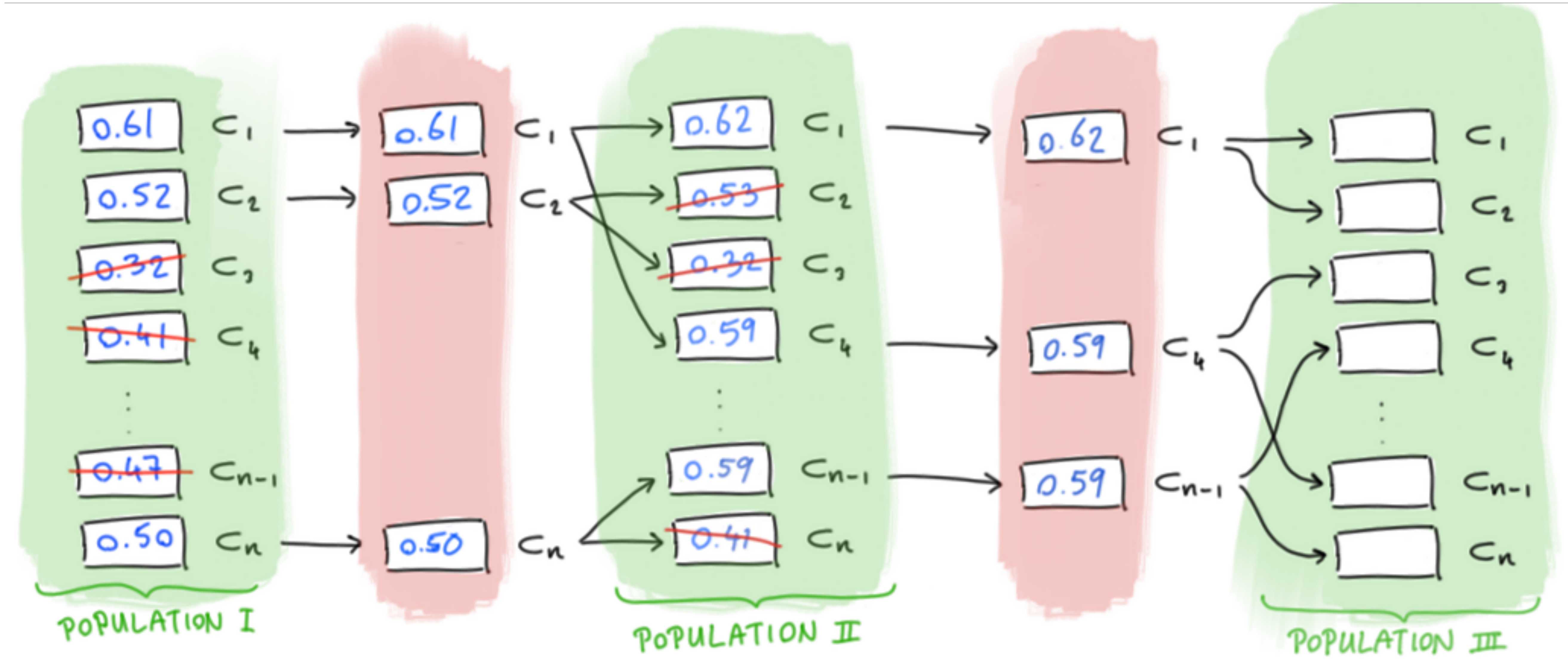
Idea of Evolutionary Heuristics.



Idea of Evolutionary Heuristics.



Idea of Evolutionary Heuristics.



Apply! Deploy!

I needed some compute power to do this, since the search space is very big. So I 'borrowed' my boss' creditcard and started planning how to scale this with my dear collegue (and engineer) Bas.

We found a solution ... it is a tool that is often marketed for other usecases ... but it ended up working very well for us.

So how to get this working in a nice way?

Despite the smaller search space, playing >2 billion games is require extreme patience. If every simulation runs for 0.1 second (which is optimistic), it would take:

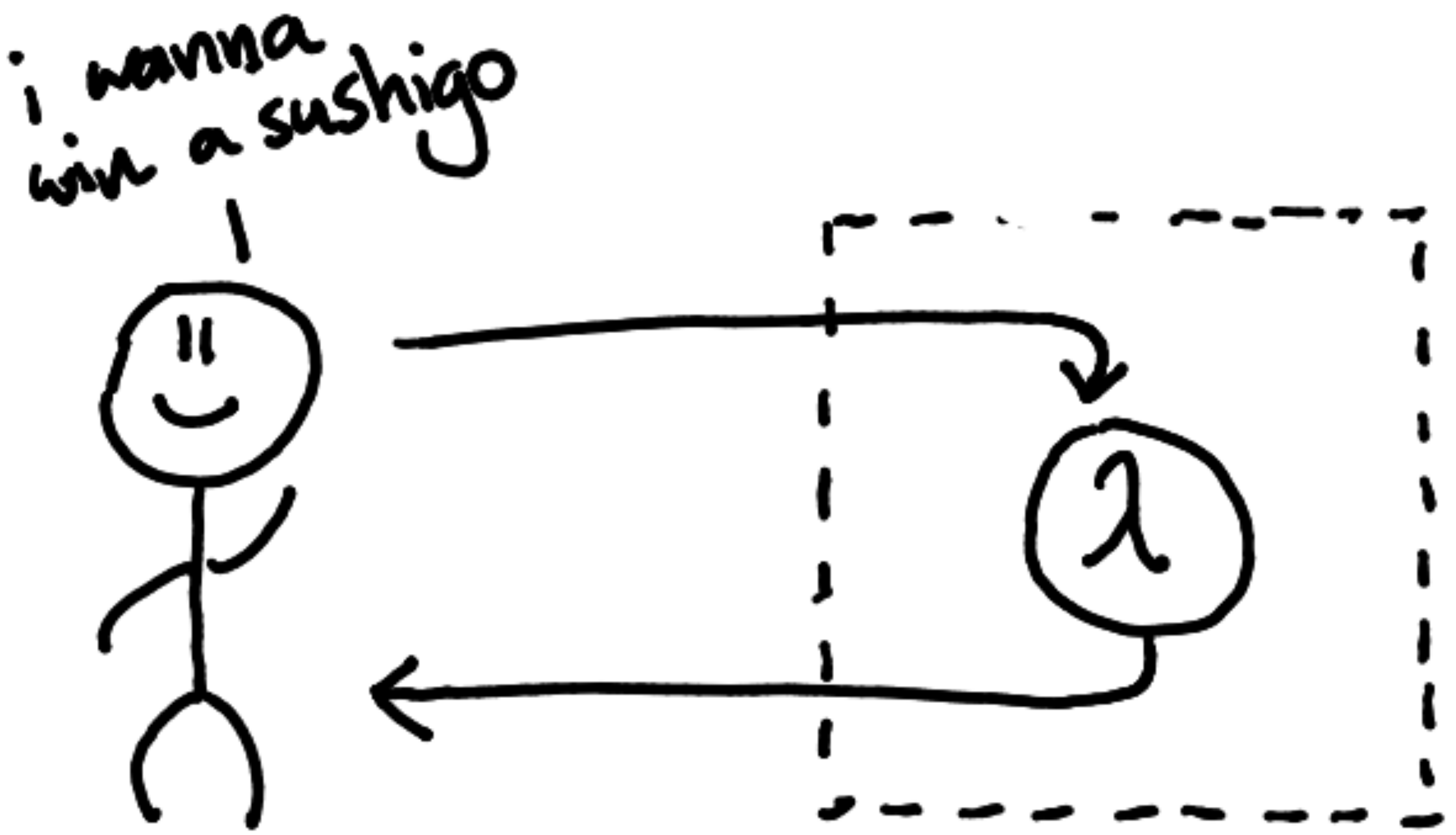
```
2_421_619_200 sims * 0.1s =
        4_036_032 minutes =
             67_267 hours =
               2_803 days = 7.68 years
```

A.I. Caramba!

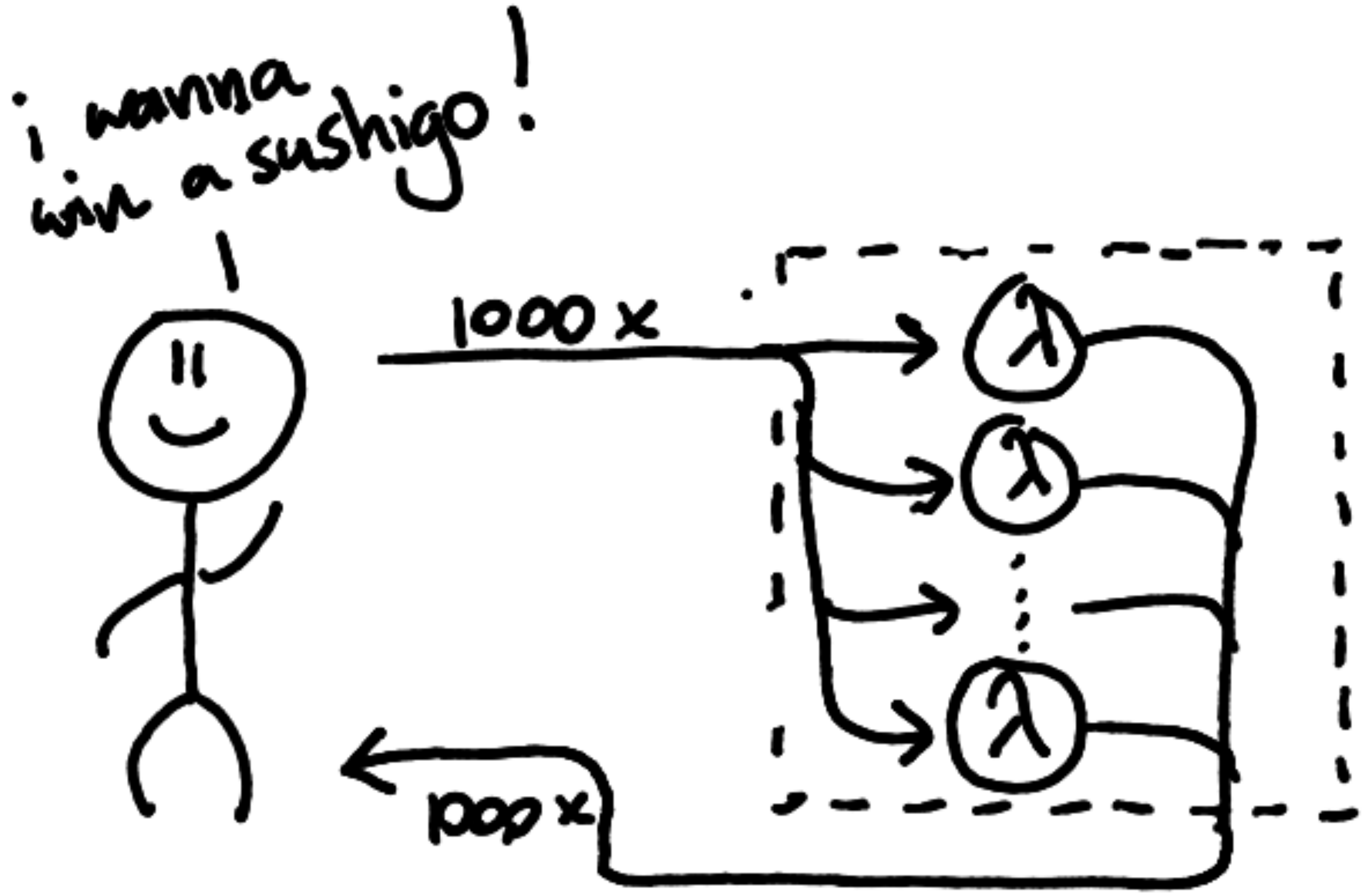
AWS Lambda

"A compute service that lets you run code without provisioning or managing servers."

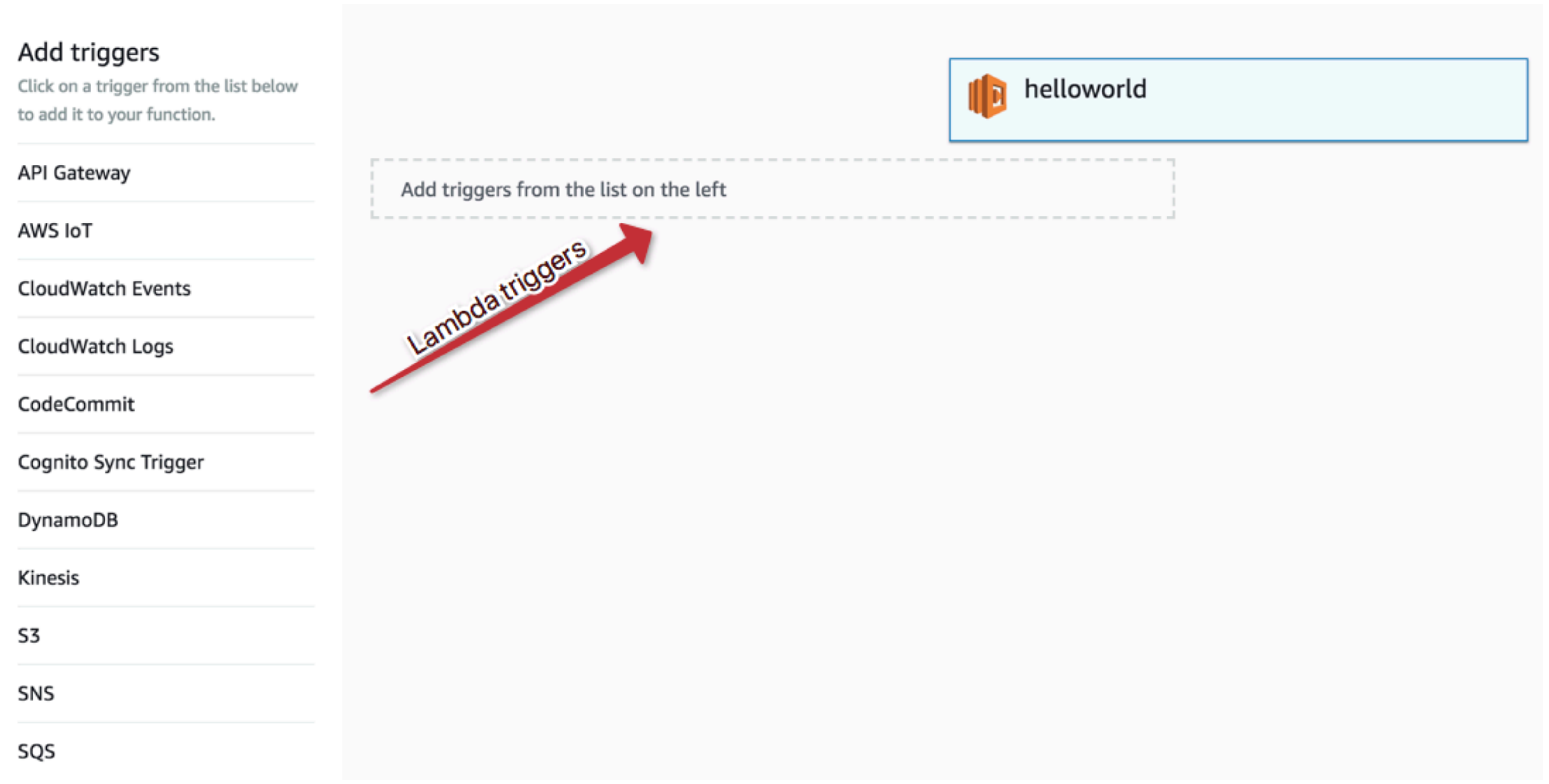
AWS Lambda



AWS Lambda



Using your Lambda

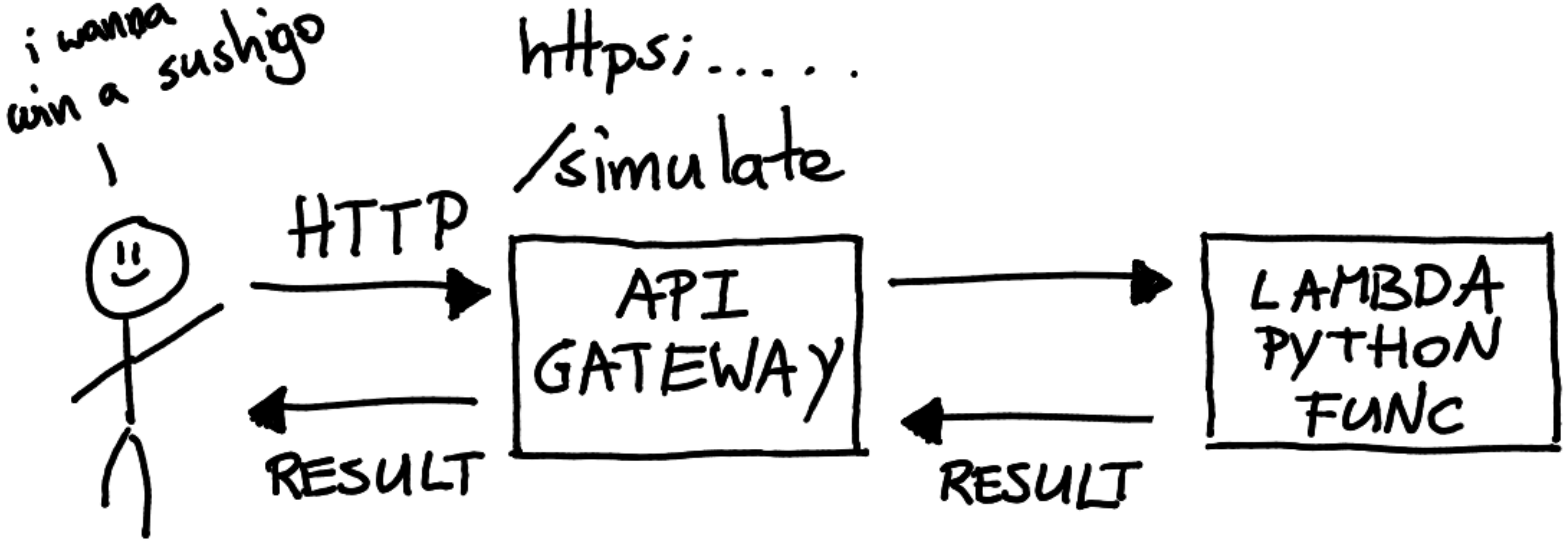


Add triggers

Click on a trigger from the list below to add it to your function.

- API Gateway
- AWS IoT
- CloudWatch Events
- CloudWatch Logs
- CodeCommit
- Cognito Sync Trigger
- DynamoDB
- Kinesis
- S3
- SNS
- SQS

AWS API Gateway



Chalice

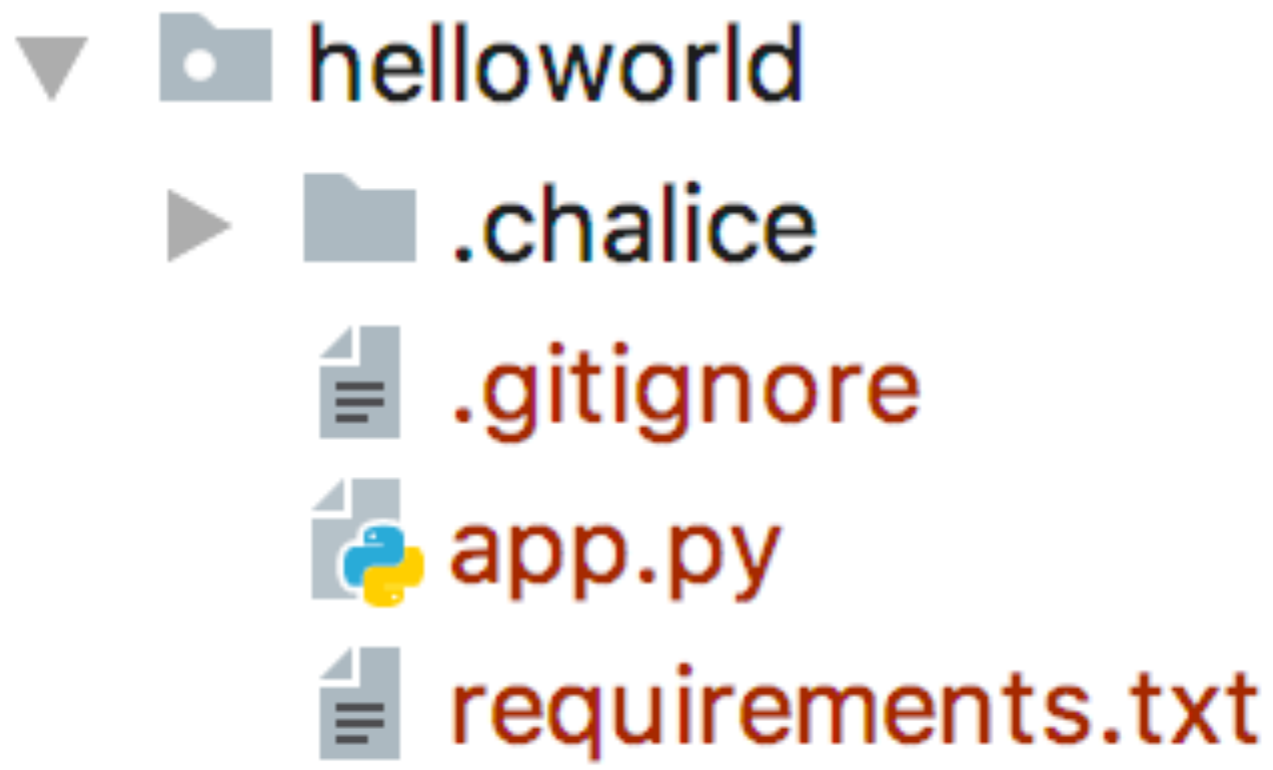
https://github.com/aws/chalice

Chalice

CLI to (super easy) create applications using AWS Lambda and API Gateway.

Chalice: Hello World

```
pip install chalice
chalice new-project helloworld
cd helloworld
```



```
helloworld
    .chalice
    .gitignore
    app.py
    requirements.txt
```

Hello world with Chalice

```
> chalice deploy

Creating deployment package.
Updating policy for IAM role: helloworld-dev
Creating lambda function: helloworld-dev
Creating Rest API
Resources deployed:
  - Lambda ARN:
    arn:aws:lambda:eu-central-1:<arn>:function:helloworld-dev
  - Rest API URL:
    https://<url>.execute-api.eu-central-1.amazonaws.com/api/
```

Lambda in Chalice code

```
from chalice import Chalice

app = Chalice(app_name='helloworld')


@app.route('/')
def index():
    return {'hello': 'world'}
```

Fire

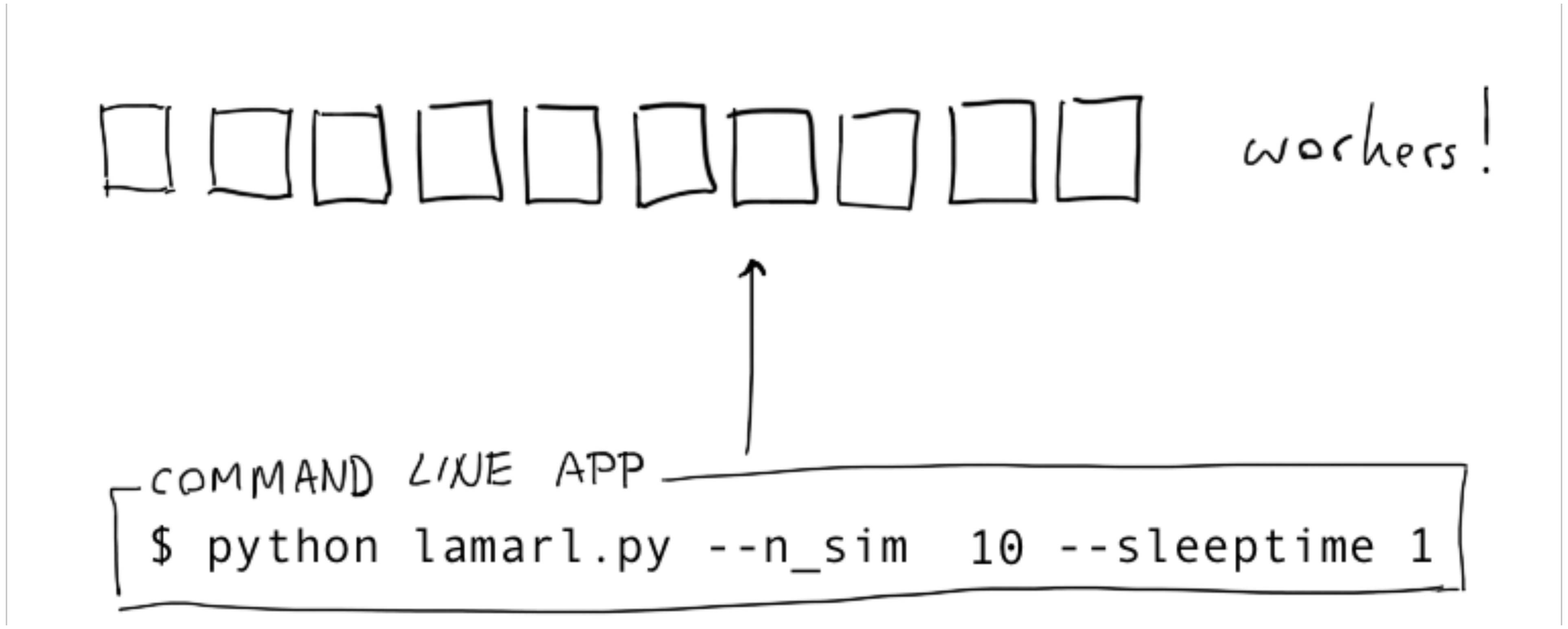
https://github.com/google/python-fire

Fire

```python
import fire

def hello(name):
    return f'Hello {name}!'

def simulate(n_sim=100, sleeptime=3):
    for _ in range(n_sim):
        time.sleep(sleeptime)

if __name__ == "__main__":
    fire.Fire({
        'hi': hello,
        'simulate': simulate
    })
```

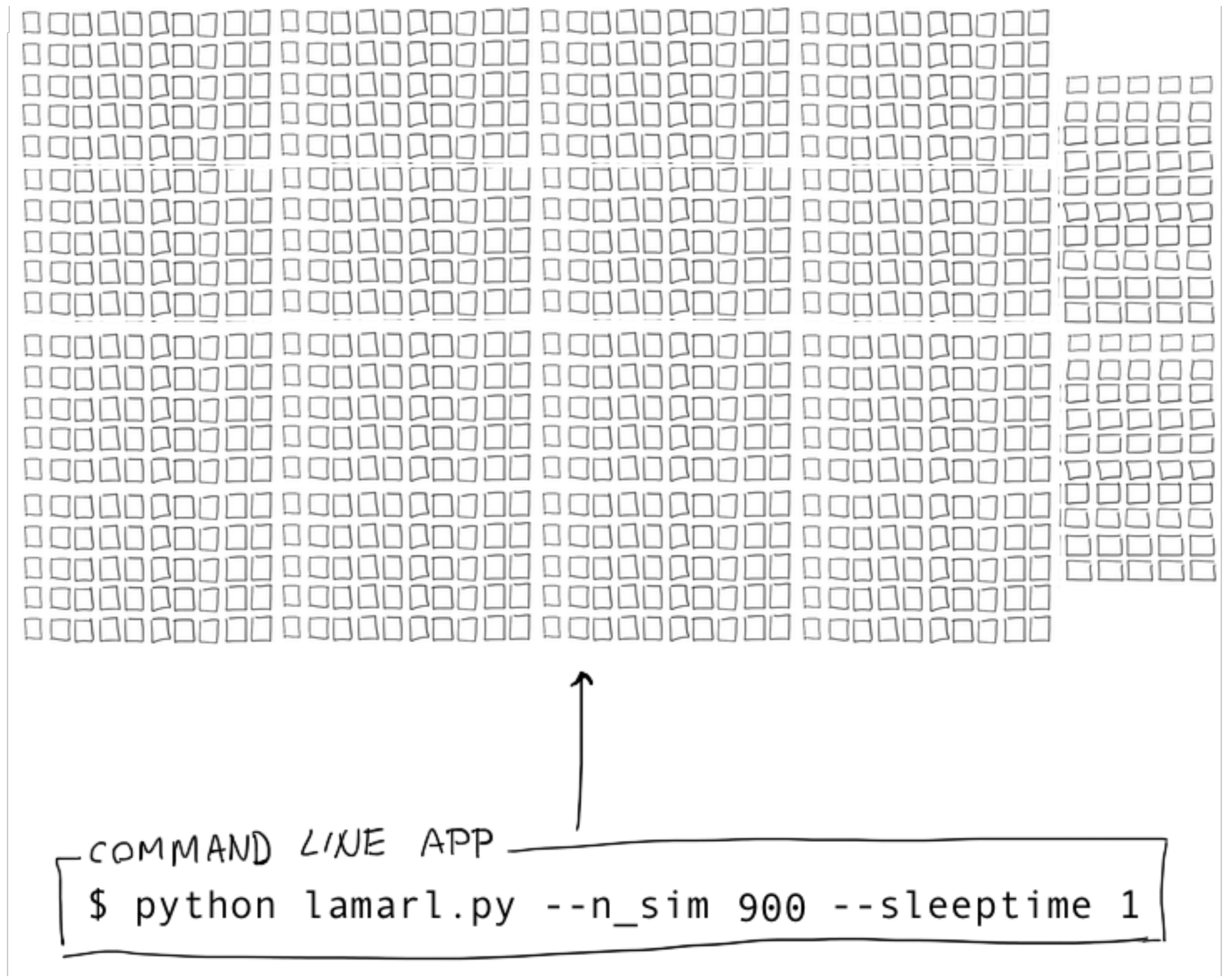
```
> python myapp.py simulate --n-sim=10
```

Command Line Apps FTW



```
$ python lamarl.py --n_sim  10 --sleeptime 1
```

Command Line Apps FTW



```
$ python lamarl.py --n_sim 900 --sleeptime 1
```

Awesome, so we have 1000 Lambdas!?

```
for i in range(2421619200):
    simulate_in_lambda()
```

Concurrency in Python

Concurrency in the stdlib:
* multiprocessing
* threading
* concurrent.features
* asyncio

Synchronous example

```python
import time


def long_task(i):
    time.sleep(1)
    print(f"Processed task {i}")


start = time.time()

for i in range(5):
    long_task(i)

end = time.time()
print(f"Completed in {end-start} sec")
```

Synchronous example

```
Processed task 0
Processed task 1
Processed task 2
Processed task 3
Processed task 4
Completed in 5.018500804901123 sec
```

Asynchronous example

```python
import asyncio
import time


async def long_task(i):
    await asyncio.sleep(1)
    print(f"Processed task {i}")


start = time.time()

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.wait([long_task(i) for i in range(5)]))

end = time.time()
print(f"Completed in {end-start} sec")

loop.close()
```

Asynchronous example

```
Processed task 0
Processed task 3
Processed task 4
Processed task 1
Processed task 2
Completed in 1.0031728744506836 sec
```

asyncio with aiohttp

```python
import aiohttp
import asyncio


async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()


async def run():
    tasks = []
    async with aiohttp.ClientSession() as session:
        for i in range(1000):
            task = asyncio.ensure_future(fetch(session, 'https://api_id.execute-api.eu-central-1.amazonaws.com/api/'))
            tasks.append(task)

        responses = await asyncio.gather(*tasks)
        # do something with responses

loop = asyncio.get_event_loop()
future = asyncio.ensure_future(run())
loop.run_until_complete(future)
```

Too many coroutines

At some point you might think about calling many many coroutines.

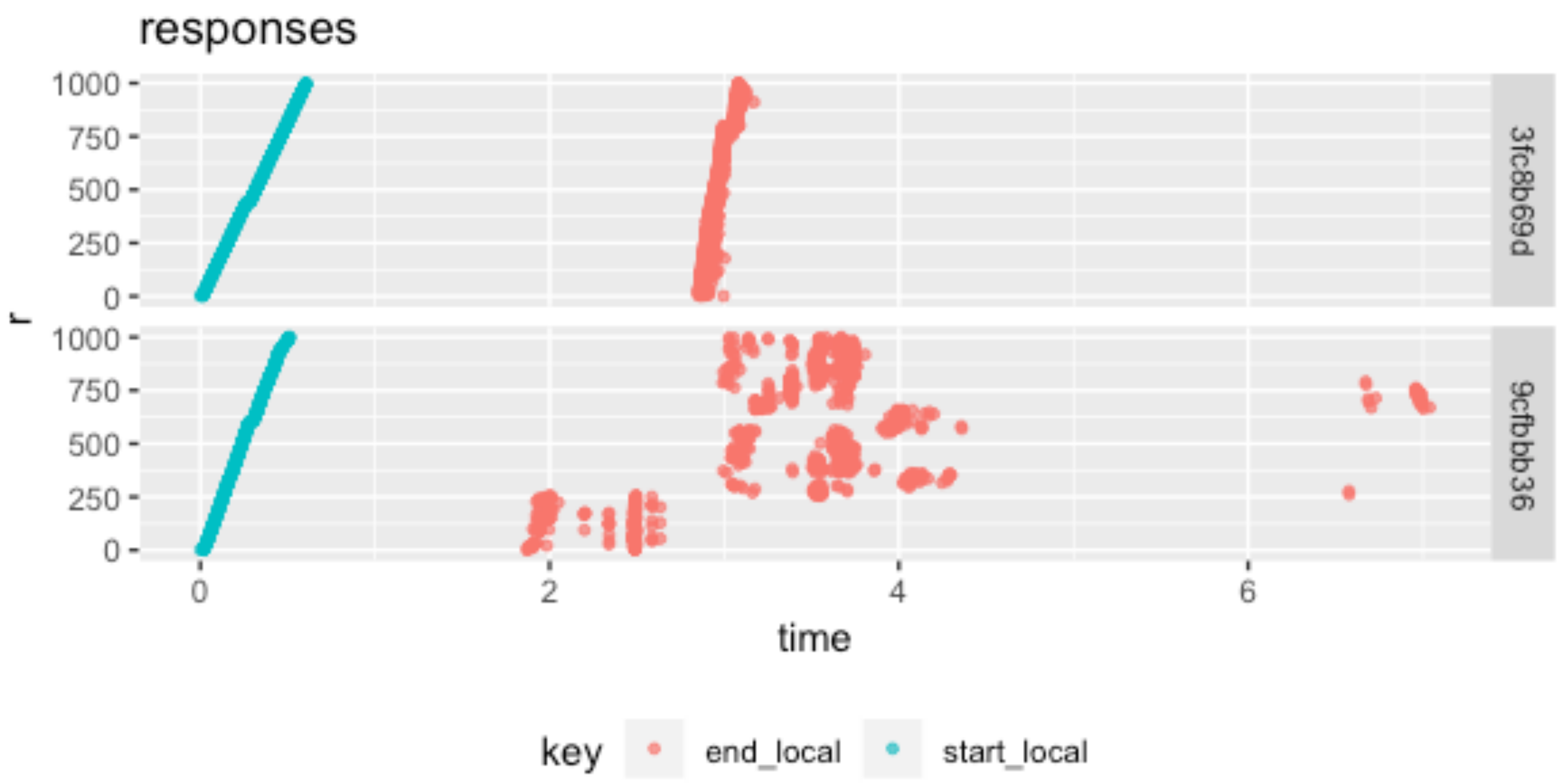
To prevent clogging up your machine, you need a Semaphore.

Time for some Experimentation

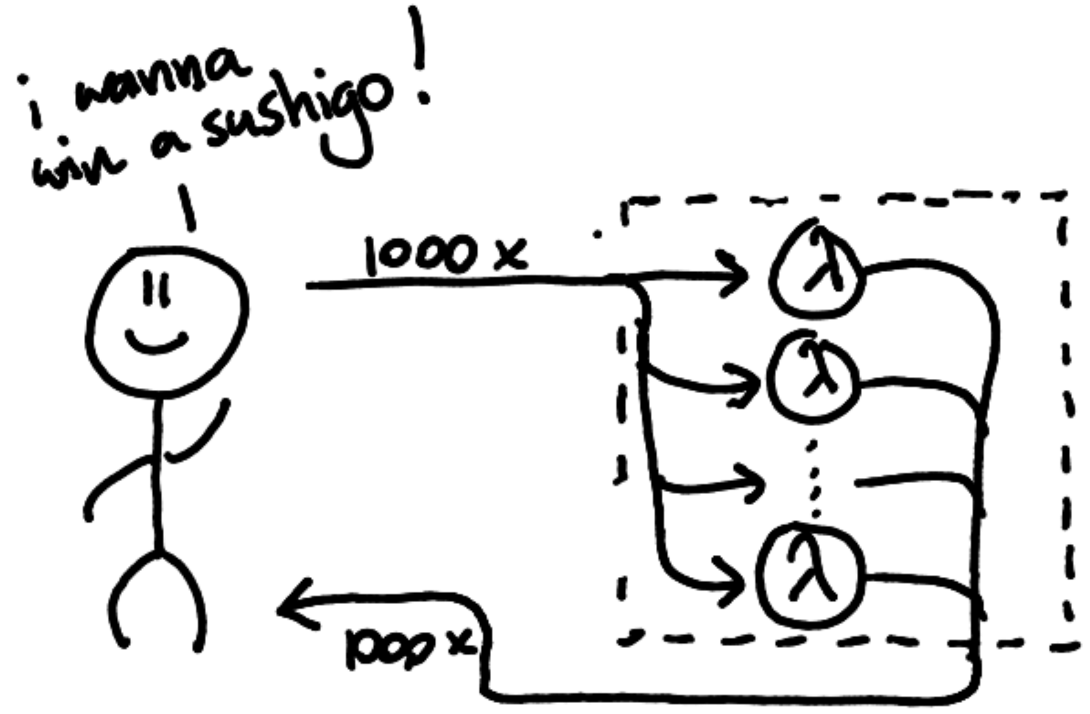
Before running the heuristic we were interested in benchmarking AWS lambda.

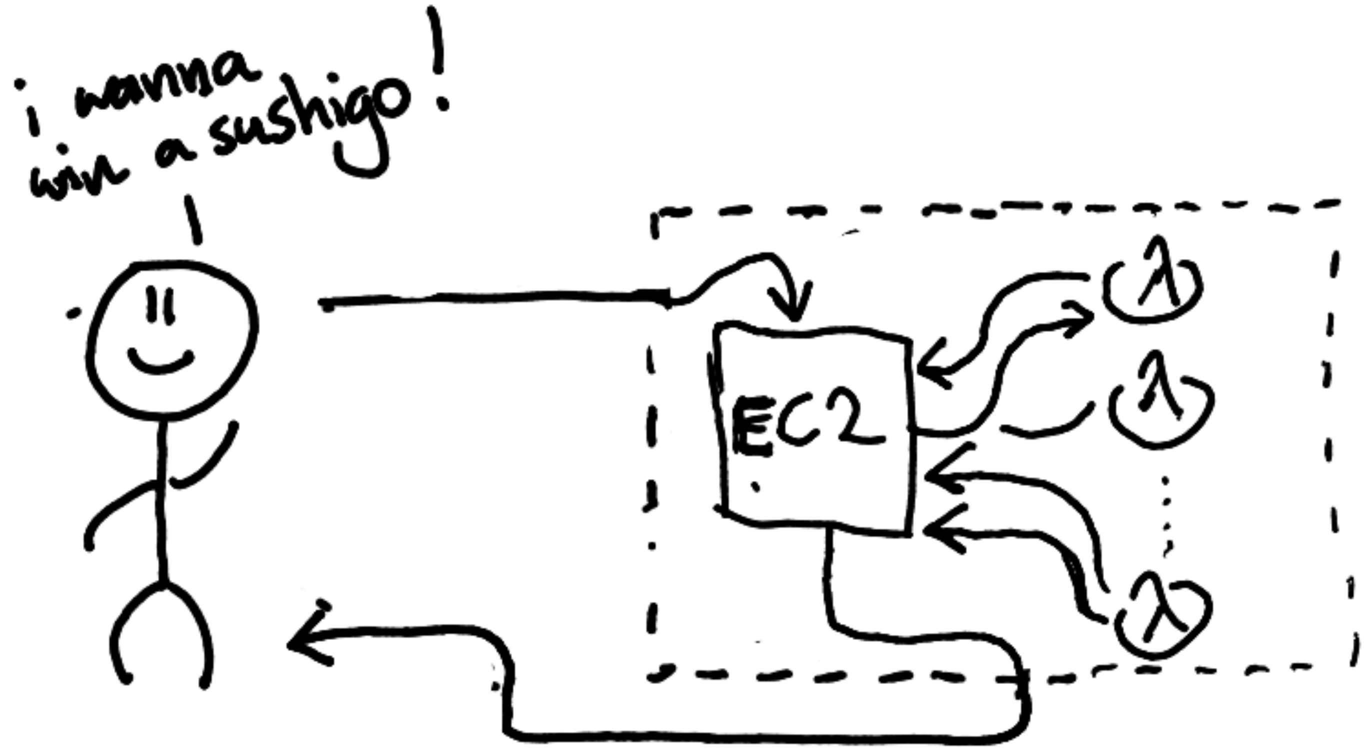
Following experiment:

- deploy a new lambda via chalice that sleeps 1.0s
- send 1000 concurrent requests from command line
- check how long it takes before everything is back


responses
r
1000
750
500
250
0
3fc8b69d
9cfbbb36
0
2
4
6
time
key
end_local
start_local

Small Experiment





Turns out, network overhead is real.

```
situation                  100       1000
hotel python3.6 wifi  2.5s      40.1s
hotel python3.7 wifi  2.5s      39.6s
aws sagemaker py36    1.3s      4.3s
```

These results suprised me, but they totally make sense.

Experimentation

```
live demo

> python command.py evolve 5 1000 5000
round: 0001/0005
aws time for round was: 20.85956 - received 1000 scores
squd,dumg,tofu,temi,wasi,tema,eggg,mak3,mak2,saln,eell,sasi,mak1,pudg
best score: 2952/5000 local-time: 0.050932s
round: 0002/0005
aws time for round was: 21.223665 - received 1000 scores
squd,mak3,mak2,temi,tofu,pudg,tema,dumg,wasi,saln,mak1,eggg,eell,sasi
best score: 2936/5000 local-time: 0.048676s
round: 0003/0005
aws time for round was: 21.3861 - received 1000 scores
squd,dumg,tofu,temi,wasi,tema,eggg,mak3,mak2,saln,eell,sasi,mak1,pudg
best score: 2960/5000 local-time: 0.045166s
...
```

Amazing!

But really, what's the speedup?

1000CPU = 1000x?

5000CPU = 5000x?

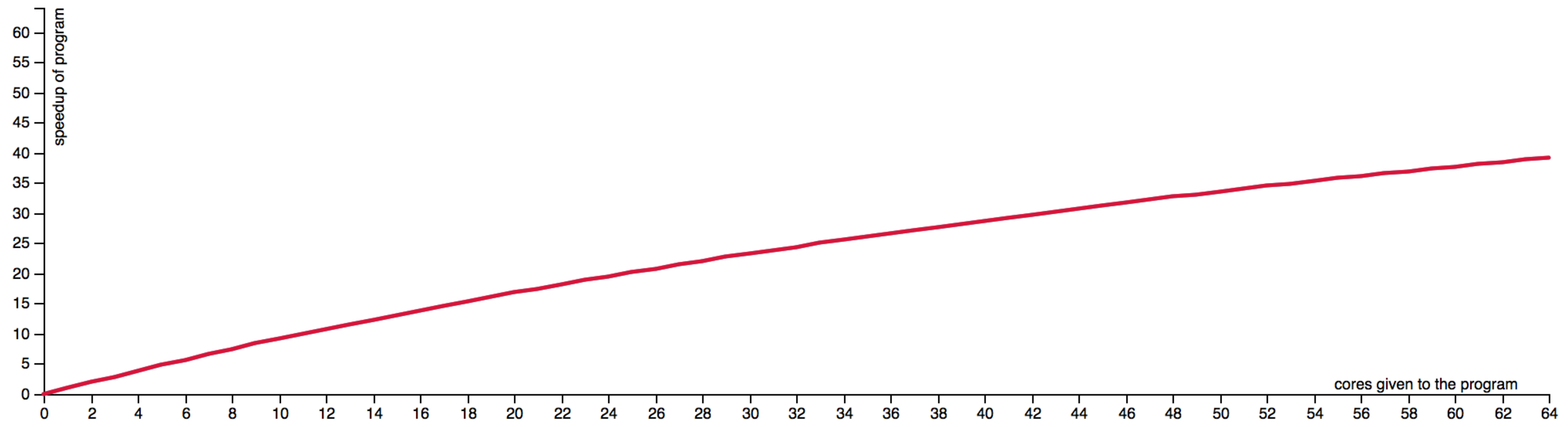
Did we really get that for free!?

Let's check the speedup

Theoretically this is impossible, partly because of **Amdahl's Law**. The law suggests if p percent of the time your program isn't running all k processes in parallel then the max theoretical speedup will be:

$$f(p,k) = \frac{\text{speedup 1 core}}{\text{speedup } k \text{ cores}} = \frac{1}{\frac{p}{1} + \frac{(1-p)}{k}}$$

Let's check the speedup



$p = 0.01$ has an speedup of $\approx 14/16$, $\approx 24/32$ and $\approx 39/64$.

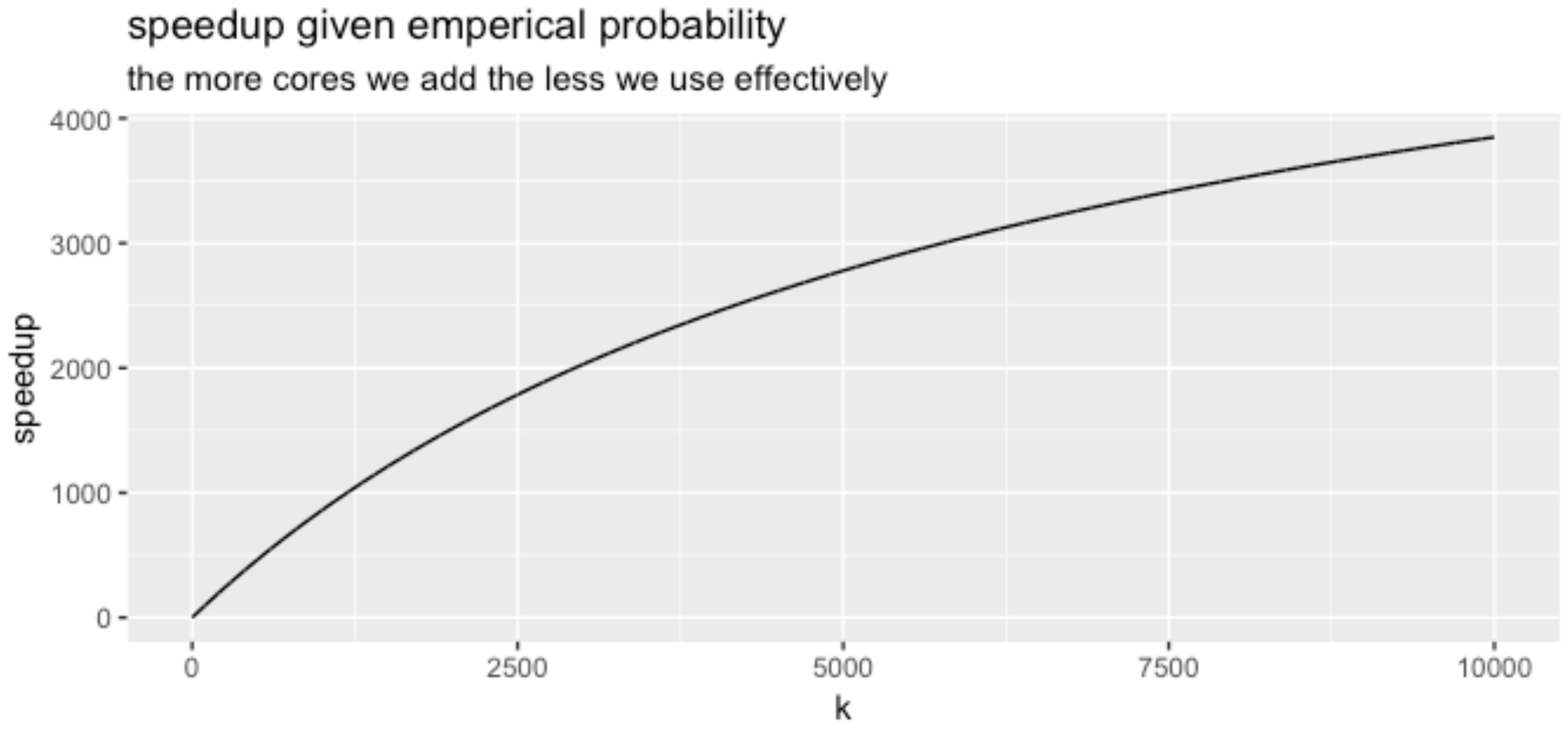
How bad was this for us?

If I just look at the total lambda time and the total local time then this seemed like a reasonable estimate for p.

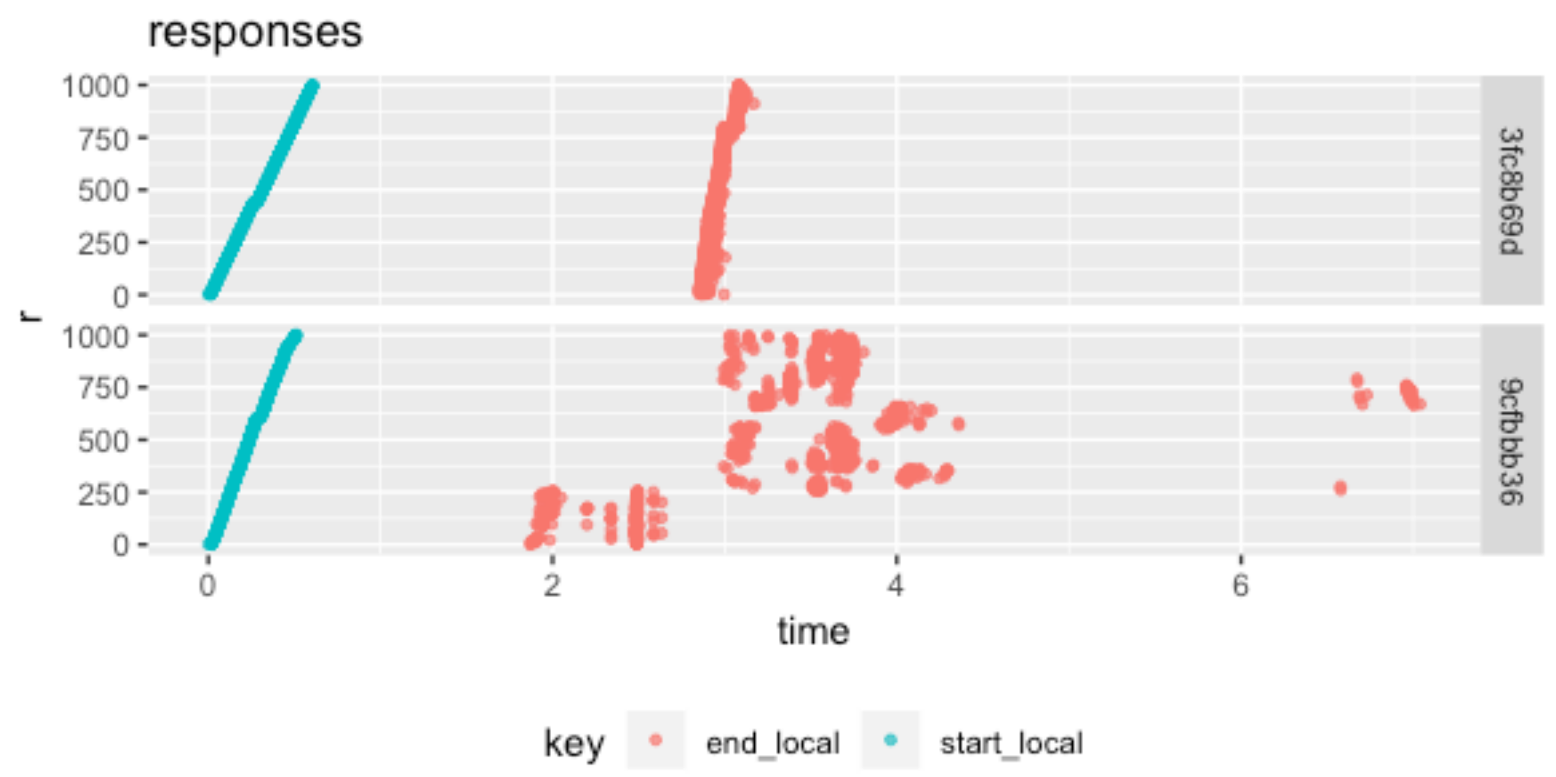
$$\hat{p} = \frac{0.000521}{0.000521 + 3.259} = 0.0001598394$$

The good news is that this number is small, but how bad will the syncing be when we run it with 1000s of cores?

How bad was this for us?



How bad was this for us?

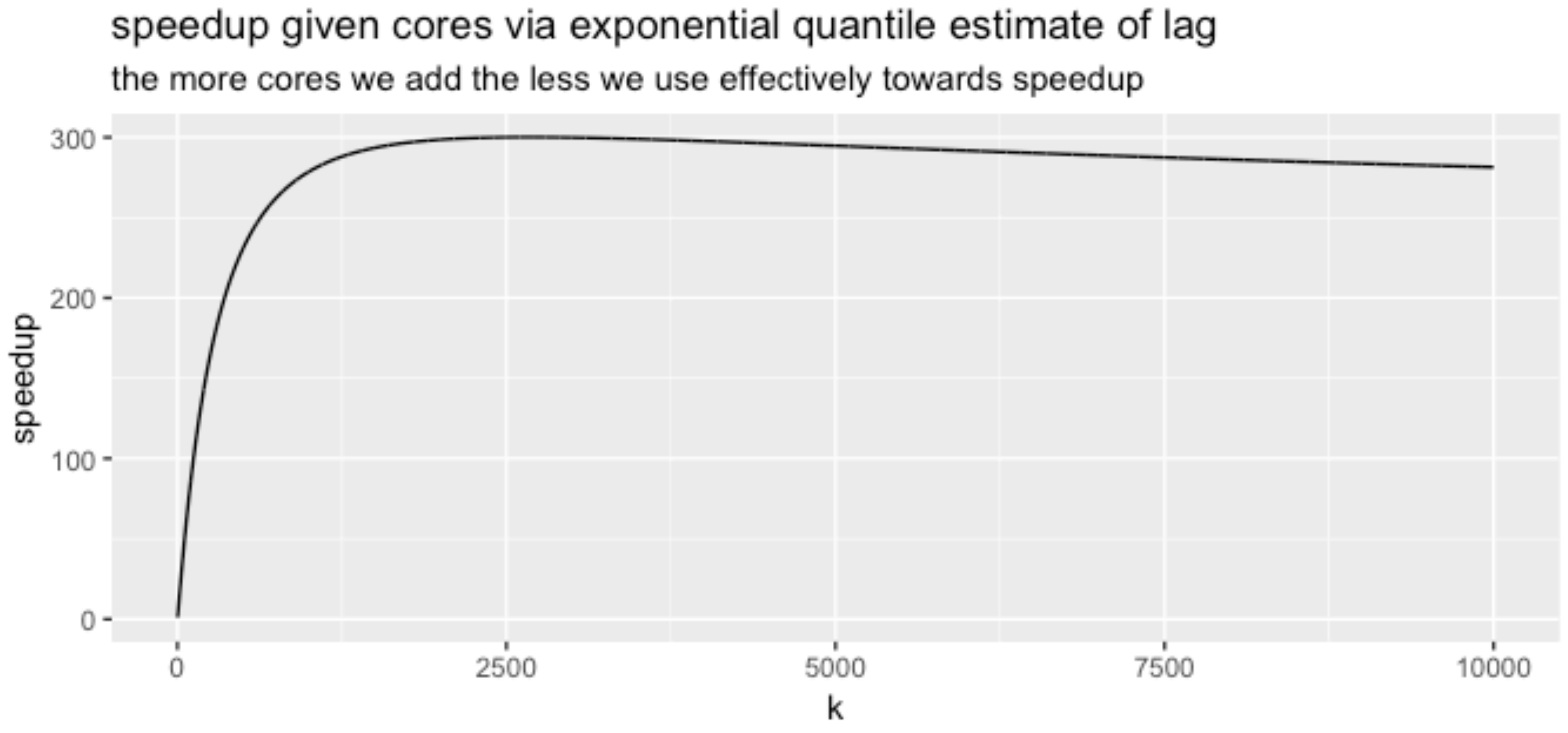


But it's **way** worse. Assuming no network overhead is silly. Even if we run it on AWS side we should assume the overhead increases as the batch size increases.

$$p^* = \frac{\text{quantile}(\frac{k}{k+1}, \exp[\lambda])}{t(\text{simulation})}$$

If we run 9000 lambda's, the waiting time for all of them is much larger than when we only run 100.

How bad was this for us?

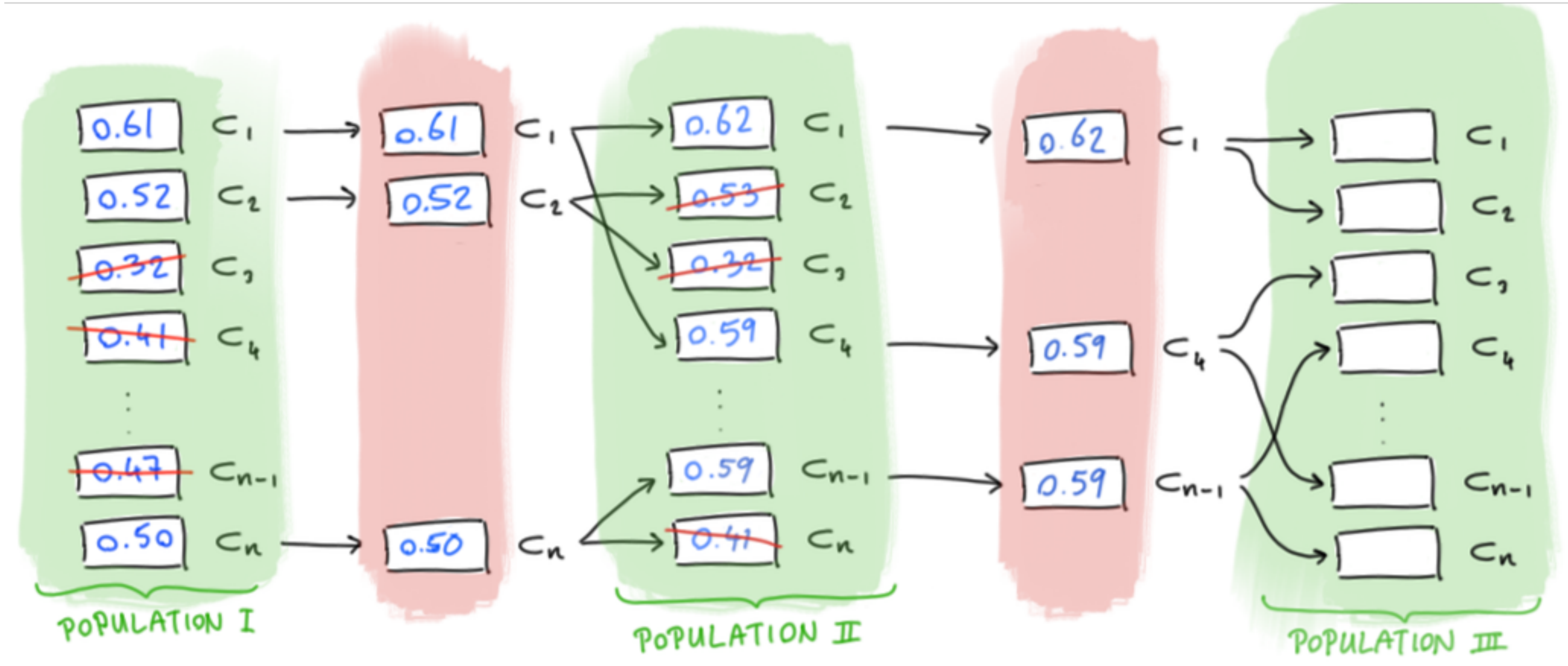


Performance improvements

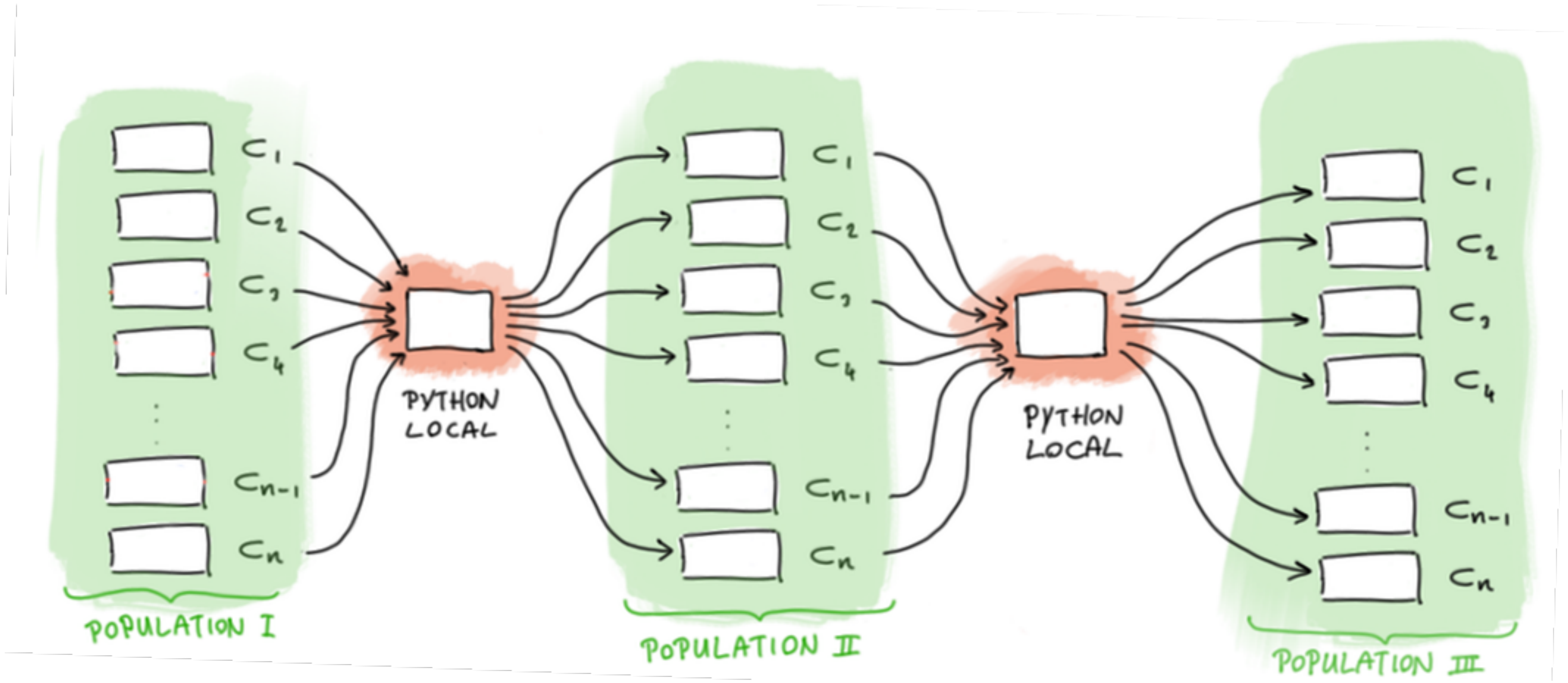
How do we make it faster? We can't just add more Lambda resources.

Interesting problem, how does one scale a genetic/heuristic algorithm? Can we finally hide our laptops?!

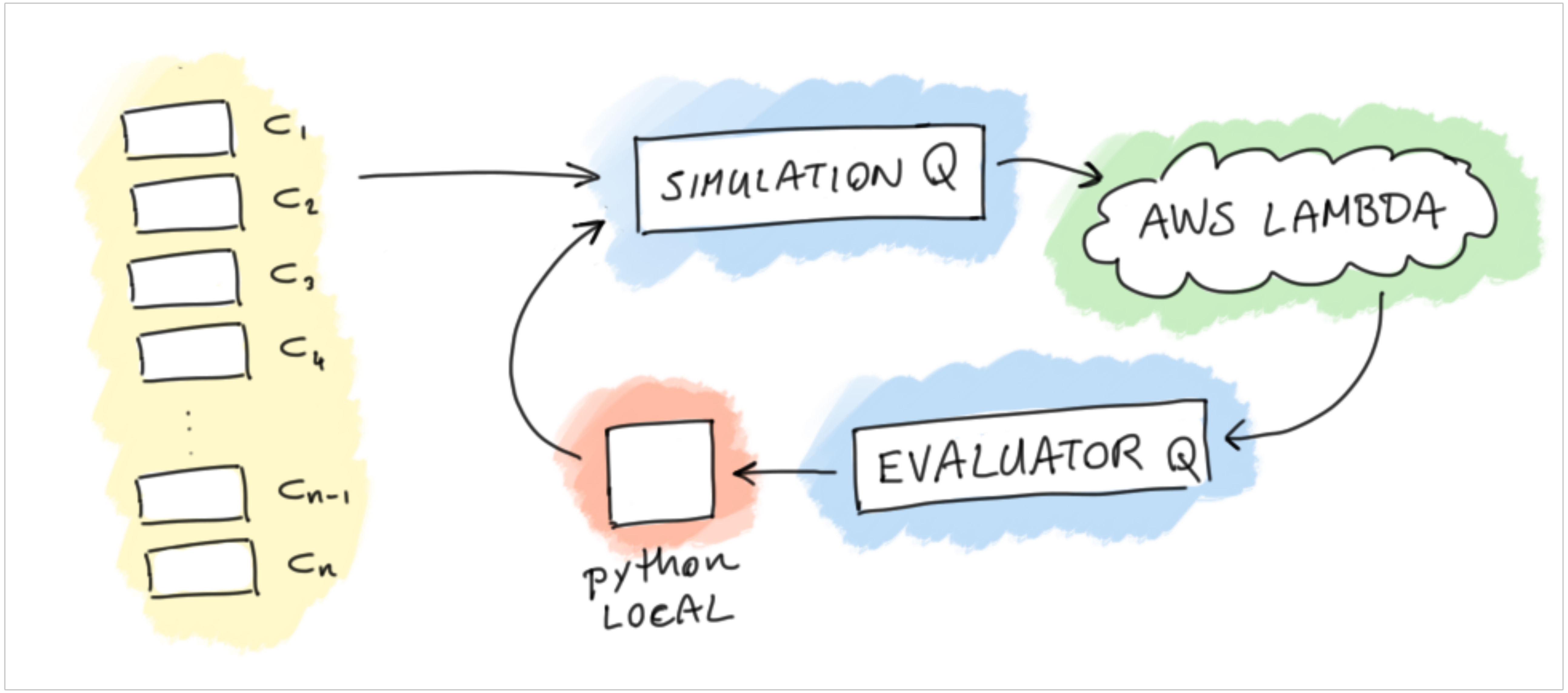
Let's check the speedup

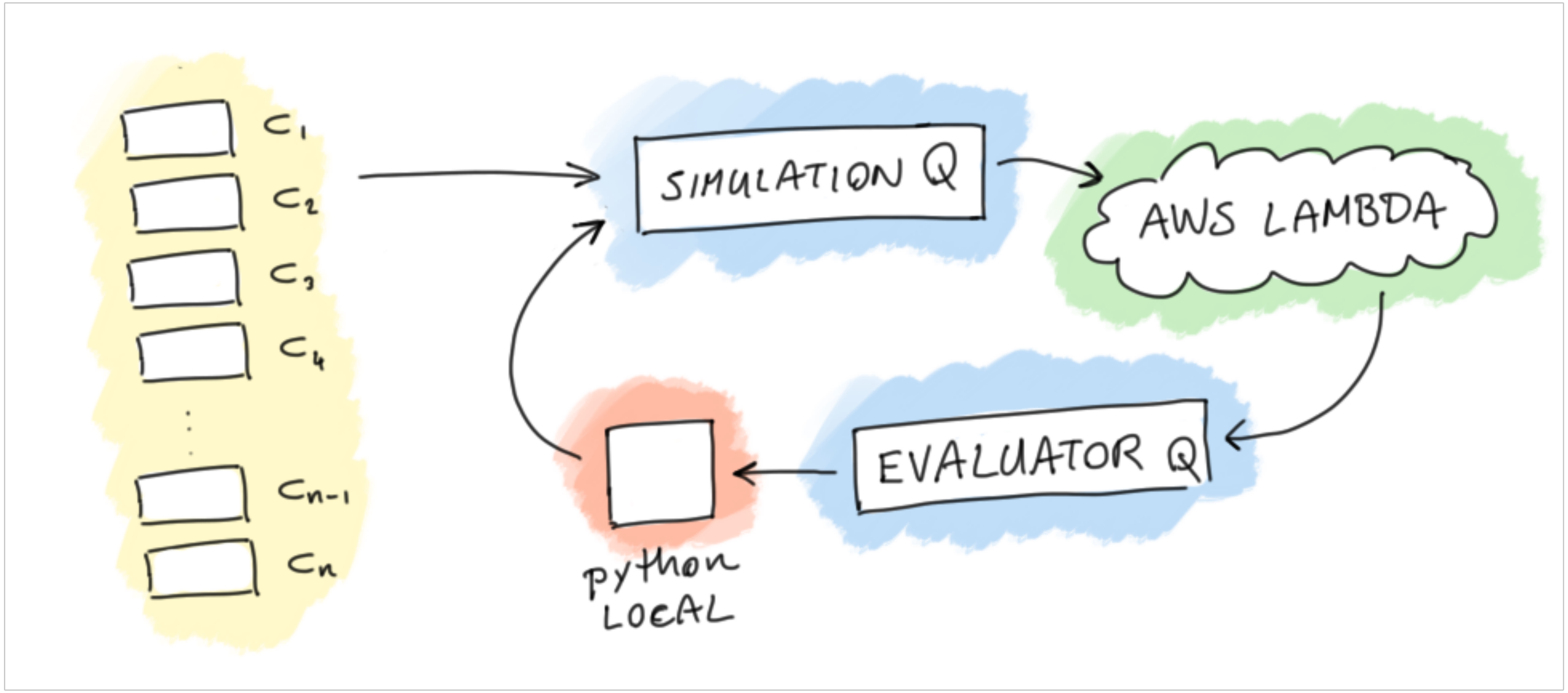


Let's check the speedup



Let's consider an alternative





The red part doesn't scale but it can handle 2000 events per second locally.

Other Improvements

There are some other improvements to be mentioned here too. We just discussed something we can do algorithmically [the maths] but there's some cloud buttons we can press too [the cloud engineering].

Chalice does a lot of work for you

Chalice is super easy and great, but Lambda over HTTP comes with limits. Our main problem was with the 30s timeout of API Gateway.

— Lambda has no direct URL

— You can call a Lambda with boto

— But boto does not do async calls

— You could try `aiobotocore` library

— But have to write the routing logic yourself

What about costs?

Lambda pricing consists of two parts:

- `0.2$` per 1M requests
- `0.000_000_208$` per 100ms of compute

Assuming a single simulation takes 10s.

$$\text{price}(\text{request}) = \frac{0.2\$}{1\text{M}} + 0.0000208 = \frac{\$21}{1\text{M}}$$

What about costs?

So that's $\approx$ 21$ per 1M requests for Lambda:

$$\mathbf{2,421,619,200\ requests = 50,854\$}$$

API Gateway pricing $\approx$ 3.5$ per 1M requests:

$$\mathbf{2,421,619,200\ requests = 8,476\$}$$

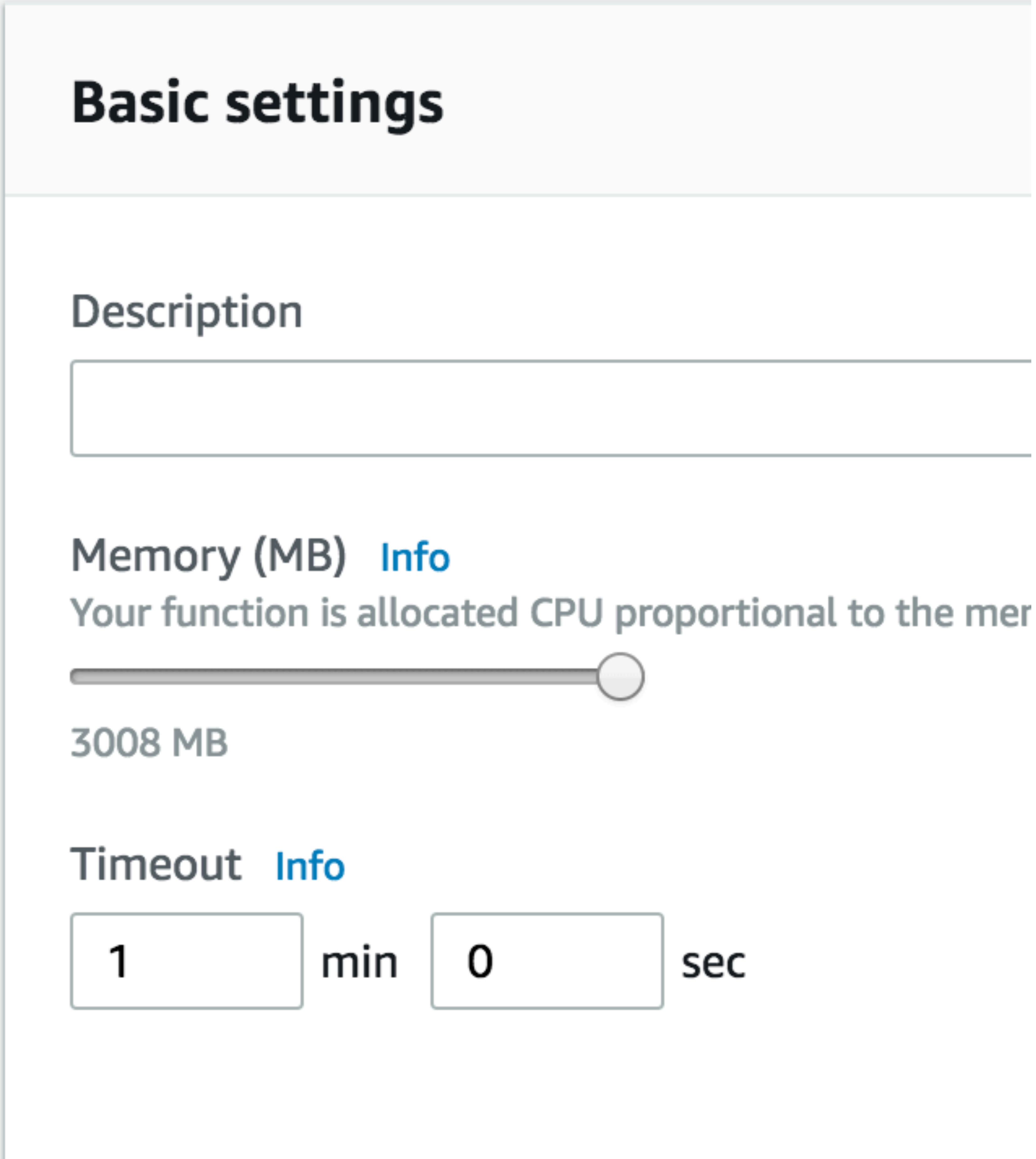
Total: 59,814$

What about costs?

If we assume no gateway and algorithm instead of brute-force.

$$\frac{21\$}{1\text{M}} \times 100 \text{ iterations} \times 5000 \text{ sims} \approx 10.50\$$$

Not too bad. For this amount of compute a 64 CPU machine needs to run for $\approx$ 24 hours (which costs 76$).



Basic settings

Description

Memory (MB) Info

Your function is allocated CPU proportional to the mer

3008 MB

Timeout Info

1 min 0 sec

What about costs?

Note that you can also optimise a bit further by upgrading the CPU of the function. It costs more per second, but the number of seconds goes down.

This makes sense because our task is very much CPU bound. If you're IO-bound, maybe don't do this.

cheapest option

```
round: 0001/0005
aws time for round was: 20.85956 - received 1000 scores
round: 0002/0005
aws time for round was: 21.223665 - received 1000 scores
round: 0003/0005
aws time for round was: 21.3861 - received 1000 scores
round: 0004/0005
aws time for round was: 20.056846 - received 1000 scores
round: 0005/0005
aws time for round was: 20.66242 - received 1000 scores
```

most expensive option

```
round: 0001/0005
aws time for round was: 5.436071 - received 1000 scores
round: 0002/0005
aws time for round was: 3.716347 - received 1000 scores
round: 0003/0005
aws time for round was: 3.610138 - received 1000 scores
round: 0004/0005
aws time for round was: 4.487011 - received 1000 scores
round: 0005/0005
aws time for round was: 3.374353 - received 1000 scores
```

What about costs?

Typically, the concurrency limit for Lambda is 1000. If you want to be able to have more functions running at the same time you need to make a request to AWS for an upgrade.

Vincent called them, explained we needed it for a card game and they gave us the upgrade within 15 minutes.

Note that if the endpoint is open on HTTP, this is a potential vector for DDOS.

Engineering Conclusions

- AWS Lambda isn't marketed for this, but our witty use of the stack actually made sense. You get a reasonable speedup for very little cost/effort.
- This is especially true for Chalice, getting started is super easy.
- There is a difference between hot/cold functions.
- State is a tricky beast.
- Async is powerful but the details are hard to get right.

Science Conclusion

Concurrency & Algorithms

It was NP-HARD

Did Vincent start winning?

Was his girlfriend in awe?

No

She even made a gif about it ...

The Truth

We ran both the batch thing and the more streaming thing (4500+ concurrent cores) and we unfortunately found out (after implementing everything and optimising a fair) that the algorithm tends to converge after only two iterations …

… in this case we had more learning from the road than from the destination.

Lessons

- The **squid** seems the best card.
- Our approach is very very naive. The game has a `rock`, `paper` and `scissors` element for example. Our approach **com-ple-tely** misses the actual gameplay.
- Never-the-less, our approach scales well and is cheap!
- Premature optimisation might be a problem still.
- We sure had fun and really learned a lot about serverless and gained a concurrenty grid-pattern that seems useful for other work.

Thanks for listening! Questions?